1994

Re-evaluating Indexing Schemes for Nested Objects

Yin-he Jiang

Xiangning Liu

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
94-023

RE-EVALUATING INDEXING SCHEMES FOR NESTED OBJECTS

Yin-he Jiang
Xiangning Liu
Bharat Bhargava

CSD TR-94-023
March 1994
(Revised 4/94)

Re-evaluating Indexing Schemes for Nested Objects*

Yin-he Jiang Xiangning Liu
Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Contents

*This research is supported in part by a grant from AIRMICS and UNISYS.

Abstract

Performance is a major issue in the acceptance of object-oriented and extended relational DBMS aiming at engineering applications. The nested index and path index schemes have been criticized for their heavy update costs and fairness for updates. This paper re-evaluates the three existing index schemes, namely nested index, path index, and multi-index for queries on nested attributes. We found that the *multi-index* is the most attractive scheme to support in an extended relational DBMS among the three. This is because the multi-index not only better balance between retrieval and update costs than the nested index and path index, but also scales well for the update cost when the number of indices increases; the *nested index* and *path index* have the update costs increasing linearly as the number of indices increases. In this paper we propose a novel design for *multi-index* that allows reusing the existing single table index structures existed in a DBMS. Our performance study complements the previous studies by extending the attributes to be *multi-valued* as well as *single-valued*. We found that a scheme of combining *nested index* and *multi-index* is a feasible solution for supporting queries on nested objects.

1 Introduction

Applications such as CAD/CAM (computer aided design/computer aided manufacturing), CASE (computer aided software engineering), and GIS (geographical information systems) etc. have the characteristics of frequent traversal of different data clusters in addition to insertion and lookup of data [1]. To better support such applications, two kinds of DBMSs have emerged. One is to integrate an object-oriented programming language (OOPL) with database technology to develop a new DBMS from scratch. This allows users to have the features of data abstraction and encapsulation, data type inheritance, and polymorphism of functions of the OOPL in addition to the full power of a programming language and capability of DBMS. The other is to extend an existing DBMS, such as relational DBMS, to support abstract data types (ADT) and inheritance to provide the equivalent power of an OODBMS. In such paradigms, abstract data types are created as *classes*. Classes form hierarchies. The objects of the same class form a *cluster*. In such database, a type of queries takes the form of selecting one cluster objects based on the evaluation of predicates on

other data clusters. Such queries are often referred to as queries involving *nested objects*, or *nested queries*.

In this paper, we investigate the index schemes for such a DBMS environment. We review the three index schemes proposed in [2], analyze their advantages and disadvantages and propose a design of the multi-index scheme that reuses the existing single table indices. We evaluate these index schemes both qualitatively and quantitatively. The discussion of our index methods is in the context of extending relational DBMS to support abstract data type. However the ideas apply to OODBMS also.

In relational DBMS, data are stored as individual relations. Connection between different relations are through join operations. Thus, an index structure usually involves just a *single* data cluster, for instance, indexing employee by salary of the employee from the same relation table. This is referred to as *single-class index* [3].

In object-oriented database systems (OODBMS) and extended relational DBMS, data are stored as individual clusters. But the connections between different clusters may be through direct link pointer, or stored query. Different objects may link together to form a composite object directly. A query on a class C can have access scope of class C and all classes linked through attributes [4]. The support of index beyond single data cluster is needed. This is parallel to the relational join index in relational DBMS.

1.1 Class hierarchy and nested structures

Figure 1 shows a class hierarchy. In the figure entities *Automobile Company Division AutoDrivetrain PistonEngine* are defined as a classes. An query on the class hierarchy is "Retrieve all automobiles made by Chrysler with four cylinders in 1991." The query consists of a predicate on the attributes *Year* and *CylinderN* of the class "Automobile", and the attribute *Name* of "Company". A type of query is to select a cluster of data based on values of another set of data. Two directions of supporting such queries have been studied by researchers. One is the nested object approach [2] and the other is that of nested relation [5]. Here we will be focusing on the nested object approach.

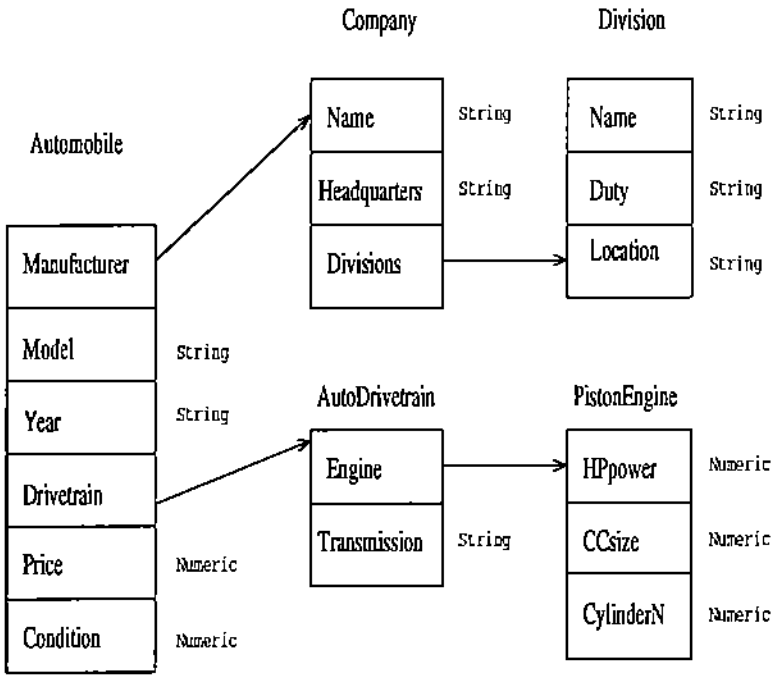

Figure 1: A class-attribute hierarchy

1.2 Related Indexing Techniques for Nested Objects

Both the OODBMS and the extended relational DBMS have the same fundamental notions of data organization – data clusters. In OODBMS the data cluster is the class of objects; in extended relational DBMS the data cluster is the relation table of objects and tuples. Thus indexing techniques applicable to one model can also be applied to another. References [6] provided preliminary discussion of the secondary indexing on a sequence nested attributes. The reference [2] proposed three index schemes and references [7, 8] access methods for nested objects in OODBMS. Our work focuses on extending the indexing organizations for nested objects proposed in [2] to better balance the retrieval and update costs. The three index schemes are called *nested index*, *path index* and *multi-index*. We refer the traditional index structures associated with single data cluster as *single cluster index*. We refer to the *nested index*, *path index* schemes as *multiple clusters indices* because the index records involve attributes of different data clusters. We illustrate the three index concepts through examples here. For detailed definitions and studies of the three index schemes

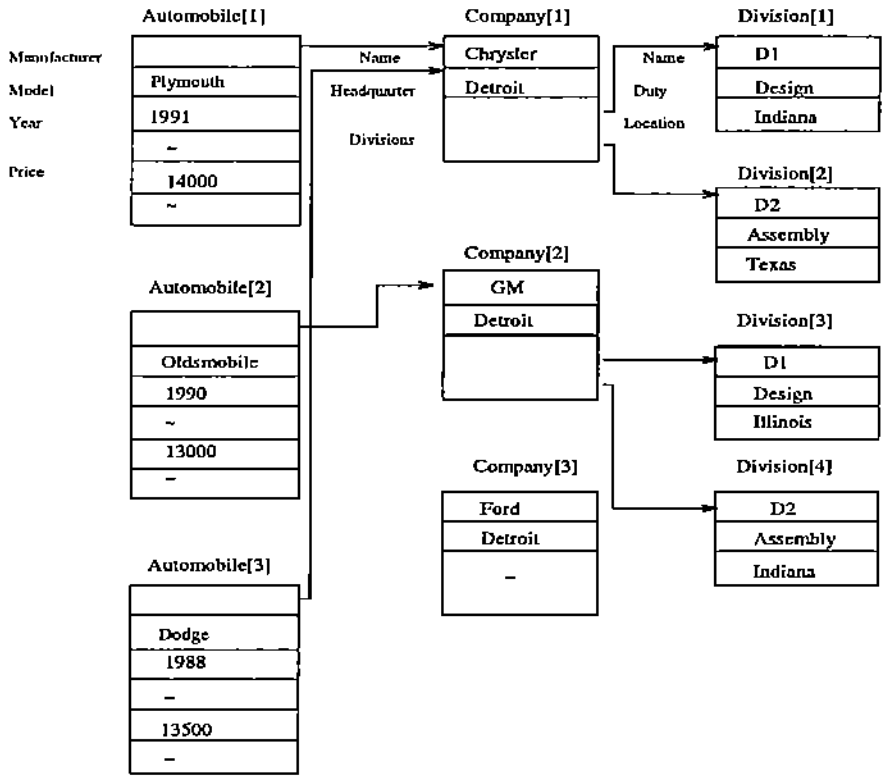

Figure 2: Instances of classes

readers should refer to [2].

Example 1 (nested index): Let us consider the class-attribute hierarchy in Figure 1. A nested index on the path $P1$ = **Automobile.Manufacturer− > Name** will associate a distinct value of the **Name** attribute with a list of object identifiers of Automobile whose **Name** is the key value. For the objects shown in Figure 2 the nested index includes the following pairs:

- $(Chrysler, \{Automobile[1], Automobile[3]\})$
- $(GM, \{Automobile[2]\})$

Example 2 (path index): Use the same example as in *Example 1* the path index will contain the following pairs:

- $(Chrysler, \{Automobile[1].Company[1], Automobile[3].Company[1]\})$

- $(GM, \{Automobile[2].Company[2]\})$
- $(Ford, \{Company[3]\})$

Example 3 (multiindex): Use the same example as in *Example 1* the multiindex will contain the two level of indexes. The first index is on **Automobile.Manufacuturer** and contains the following pairs:

- $(Company[1], \{Automobile[1], Automobile[3]\})$
- $(Company[2], \{Automobile[2]\})$

The second level index is on **Company.Name** and contains the following pairs:

- $(Chrysler, \{Company[1]\})$
- $(GM, \{Company[2]\})$
- $(Ford, \{Company[3]\})$

The indices for nested objects aim to speed up the queries for selecting data objects based on the values of other objects pointed to by the selected objects. For instance in Figure 3 without the indices for nested objects a query processor has to follow a chain of pointers from an object A to the target object B, evaluate a predicate on object B, based on the predicate value select or reject object A. Such a process has to be done for every object in a cluster of A. With the indices on nested objects from cluster B to cluster A, the previous select query can be processed as first consult the predicate on the index, then select objects out of cluster A based on the index values returned. The single table indices can not be of any direct help in the queries involving nested attributes, because they are involved only with one cluster of data.

1.3 Pros and Cons of the Multiple Clusters Indices

The nested index speeds up retrieval queries involving only attributes of the end classes on the path. The nested index requires the support of both forward and backward traversal of objects during updates of values of the middle objects in the path. The path index accelerates retrieval

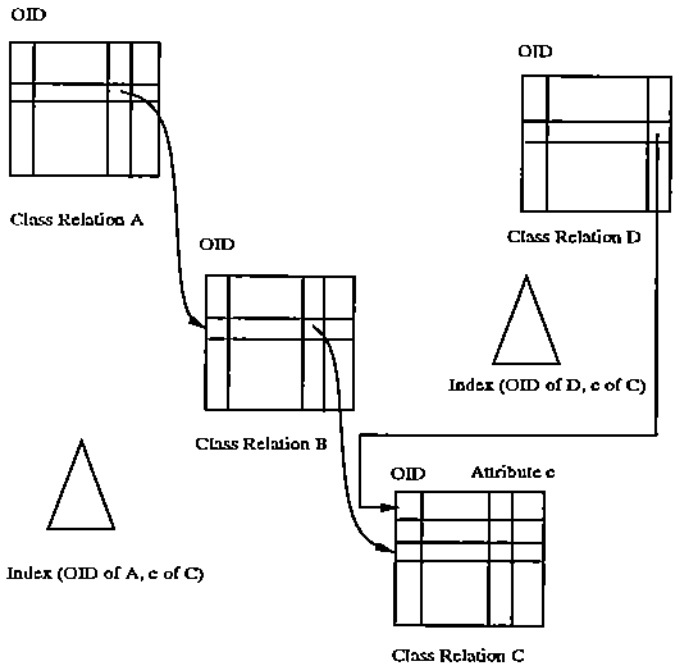

Figure 3: The Nested Index Proposed by Bertino et. al

queries on two attributes along a path. The path index requires the support of forward traversal of objects during updates of the middle objects in the path because it stores the whole path in the leave node of the B-tree index. The multi-index speeds up retrieval queries involving any pair of attributes along the path. Thus the applicable scope of the multi-index is the super set of those of nested index and path index. Furthermore it requires neither forward or backward traversal of objects.

The degree of retrieval speedup and update overheads vary among the three schemes. As pointed out in [2] the most important parameter affecting the performance of indices is the degree of reference sharing among objects for the various classes in the path. As for retrieval performance the nested index has the best retrieval among the three schemes, followed by path index and the multi-index. However, the retrieval performance of the multi-index is independent of the index record size and increases linearly with the product of degrees of reference sharing. As for update performance the multi-index is the best. For path length of two the nested index has a slightly

lower cost than the path index. For path length of three, if the updates are primarily on the first and second classes, the nested index has a slightly lower cost than the path index; if the updates are largely on the last class on the path, the path index has significantly less cost than the path index.

The nested and path indices have several significant disadvantages for update costs.

- The number of indices for nested objects associated with a cluster of data may be indefinite. This is because there may be many different paths that lead to a cluster. And each path may need an index for nested objects.
- An update including insert data to a cluster may involve updates to *indefinite* number indices for nested objects associated with the cluster.

Updates to either of the linked clusters may result in updates to the index structures. A cluster can be pointed by multiple clusters, each reference link can have nested indices built. An update to this based cluster can result in updating the multiple index structures. It is stated in [9] that the O_2 object-oriented database system does not support such path indices for nested objects because it is unfair to the update users for maintaining indices for all the paths. For updates, the multi-index performs better than the nested index and path index in that no forward or backward traversal of objects required and only one level of index is updated.

2 Indices for Nested Objects

Our goal is to use the existing single cluster indices to achieve the similar effect of the multiple clusters indices. The motivation is that single cluster indices allows a much better update performance than the multiple cluster indices. This is because the single cluster index has been incorporated by all database systems and updates costs for those indices always have to be paid. The retrieval performance will be better than the case without any indices because the use of index structures avoids the linear search of data. Our design of the *multi-index* is based on the following observation:

- The the *nested index* and the *path index* are skewed heavily towards retrieval performance at the expense of update costs.

- The multi-index achieves better balance between retrieval and update performance than the nested index and the path index by partitioning the path into multiple levels. Each level of a multi-index is a nested index which involves a pair of attributes from different classes.

- The object identifiers (OID's) are constant during the life span of objects and other members of a class can be modified. For instance tuple identifiers change as tuples are deleted and inserted.

- Single cluster index to map any attribute values in the cluster into a set of object instances. In relational DBMS the single cluster index maps an attribute to the tuple identifiers of a relation. In OODBMS and extended relational DBMS a single cluster index maps one attribute member to the objects identifiers (OID's) of a cluster.

- In OODBMS and extended relational DBMS the data linking is usually implemented as storing OID's of the pointed objects in the attributes of the pointing objects. We can maintain the linking information between data clusters by pairing the OID's of the linked objects.

We design the *multi-index* for nested objects using only *single cluster indices*. It consists of two parts. One part is the single cluster index structures for each cluster mapping an attribute to the OID's of objects in the cluster. The other is the index structure on the pointing cluster mapping the OID's of the pointed objects to OID's of pointing objects. This is still a single cluster index because in our extended relational DBMS, the pointer attribute contains the OID of a pointed object as a component. The mapping of OID's between different clusters can be realized by hash index or B-tree index. The design allows a relational database system's index structures to be reused in processing the nested objects queries without any modification. Compared to the path indices our approach is fair to the users updating data. Only when updates are made to an indexed cluster, do users pay the overheads to updating that one index structures.

Note that multiple cluster indices are additional structures to the existing single cluster index structures. Special mechanisms are needed to recognize and maintain the multi-index structures. Our design of multi-index also requires a mechanism to recognize the existence of the multi-index. However the maintenance of the multi-index is done by the DBMS because all the DBMSs maintain

their single cluster index upon updates on attributes including pointer attributes. Thus our scheme has the advantages of smaller storage costs and update costs because of the reuse of the existing single cluster index.

The Multi-Index Structure The index structures are conventional B-tree index structures on simple type values such as integers and character strings [10]. There are two types of leaf record. One is (Value, {OID's}) where "value" is one attribute value and {OID's} is a set of OID's whose objects have that value; the other is (OID, {OID's}) where OID is that of the pointed object and {OID's} is the set of OID's whose objects pointing to that OID object.

The multi-index requires two levels of support. One is in the Data Definition Language (DDL) to specify on which members of classes a multi-index is built. The other is in the Data Manipulation Language (DML) for recognizing the existence of multi-index during processing a select query. The maintenance of a multi-index is the same as single cluster index, thus is taken care of by a DBMS automatically and no additional efforts needed for query processor.

Multi-index Operations

Retrieval With the multi-index the evaluation of the retrieval queries involving nested objects becomes a *backward* evaluation. The multi-index can be described using the Figure 4. Suppose that we select objects from cluster A, based on the value of member values c of the pointed object in cluster C.

- Using the simple index structure for member c of cluster C, we acquire a set of OID's for cluster C that satisfy the predicate.
- From this set of OID's, we find the set of OID's of objects in cluster B that link to those objects on cluster C by doing the index read on the OID field of the pointer attribute in cluster B.
- We can propagate this backward evaluating process until we get the set of OID's of the required cluster A.

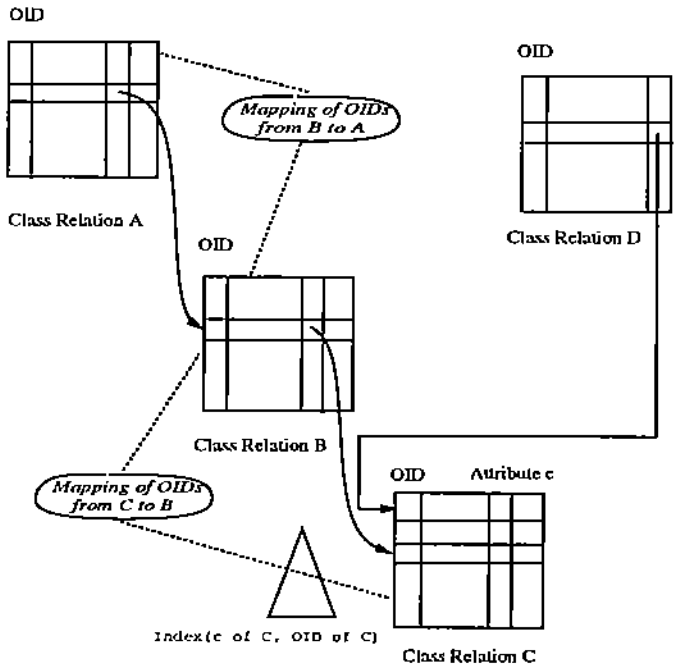

Figure 4: The Index Organization that Adapts to the Nested Objects

- We then perform an index read on OID field of the cluster A to retrieve all the objects with OID's in the set. Project the required attributes of the objects.

The most frequent operation in the multi-index scheme is that *submit a set of values to a single cluster index and acquire another set of values.* There are two ways to perform this operation. We can always submit one value at a time until exhausting the set. It seems more efficient to submit a set of OID's once to perform the index lookup especially if index lookup is handled remotely.

Here is the algorithm for processing a retrieval query.

- Find from a query its projected attributes in the predicate clause and their corresponding clusters.

- Find the path that connect these two clusters. Check the meta data for the existence of a multi-index for the path.

- If there is an index built on the path search the index structure and return those objects satisfying the predicate as described in the previous example of Figure 4.
- Otherwise follow the pointers to search every object in the path and evaluate the predicate.

We use the example query

```
select g.pStudent->name from grad_students
   where g.pAdvisor->grant > 100,000
```

Processing a select query using nested index can be done as follows:

- Evaluate the predicate using single table index. In our case use the B-tree index on "grant" of the relation "Professor" to obtain a set of OID's of professors.
- Map the OID's obtained from last step into a set of OID's of the projected attributes. In our example we acquire a new set of OID's of "grad_students"
- Based on the set of OID's retrieve the objects of "grad_students'. Project the "name" attribute of the set of "grad_students".

Update Updates to the attributes of a path have two kinds. One is the update of pure values of an attribute in the end cluster of a path. The other is the update of the links between objects. For the multi-index both of these updates are single cluster index updates. For extended relational DBMS such single cluster indices are maintained by the underlying relational DBMS. No forward and backward traversal of objects are required during the update of the multi-index as opposed to updates of path and nested index. No special actions need to be taken by the query processor for updates to the linkage of objects for the multi-index as opposed to the multi-index mechanisms are needed to recognize the existence of multi-index and issue an update the index from the query processor. In the next section we compare the performance of the multi-index and multiple clusters indices in more details.

3 Performance Evaluation

In section 1.3 we summarize the comparison studies of the nested index, path index and multi-index presented in [2]. The reference [2] considers the cost of multiple clusters indices under a single path for a cluster and the path length of two. To further compare the multi-index with the multiple cluster indices we consider both single path and multiple paths existing for a cluster. We also consider the path length of three. For path length of more than three, as pointed in reference [2], it is general preferable to split the path in several subpaths of lengths one, two, or three. We use number of disk pages as the measure for the storage, retrieval and update costs. The organization of the rest of this section is as follows. First we state the parameters and assumptions used. We then study the storage, retrieval and update costs in turn. For each part we justify our computation and study the case for path length of three for single-value attributes, followed by the case for *multiple-valued* attributes. Our study here complements the study of [2] for evaluating the index schemes for queries on nested objects.

3.1 Parameters of the Index Cost Model

We use the analytical model and parameters introduced by Elisa Bertino and Won Kim in [2] and list them as follows:

Given a path $\wp = C(1).A(1).A(2)...A(n)$, the following parameters that describe the characteristics of the classes and attributes.

Logical Data Parameters
$D(i)$: Number of distinct values for attribute A(i), $1 \leq i \leq n$. In particular, when $1 \leq i < n$, this parameter defines the number of distinct references from instances of class $C(i)$ to instances of class $C(i+1)$ through attribute $A(i)$.
$N(i)$: The number of instances of class $C(i)$. $1 \leq i \leq n$.
$k(i)$: Average number of instances of class $C(i)$ with the same value of $A(i)$. As pointed out in [4] for single-value attribute, $k(i) = \lceil N(i)/D(i) \rceil$. For "Student", "Professor" clusters a query on nested attributes is that "select students whose advisors have grants of certain amounts." In the example of $C(1)$ is "Student" class and $A(1)$ is "Professor" class, $D(1)$ is the number of Professors; $N(1)$ is the number of ; Students and $k(1) = N(1)/D(1)$ is the average number of students sharing the same professors.
$OIDL$: Length of the object identifier in bytes.

System Parameters
P: Disk page size.
pp: Length of a page pointer.
IO: I/O time to fetch a disk page.

Index Parameters
d: Number of keys in a nonleaf node.
f: Average fanout from a nonleaf node. $d \leq f \leq 2d$, for nonleaf node other than the root. $2 \leq f \leq 2d$, for the root node.
kl: Average length of a key value for the indexed attribute (i.e., $A(n)$).
kll: Size of the key-length field.
rl: Size of the record-length field.
$noid$: Size of the number of oids ($n.path$) field.
ol: sum of kll, rl, $noid$.
L: Average length of a nonleaf node index record. $L = kl + kll + pp$.
DS: Length of the directory at the beginning of the record, when the record size is greater than the page size.
XN: Average length of a leaf-node index record for a nested index.
XP: Average length of a leaf-node index record for a path index.
$XM(i)$: Average length of a leaf-node index record for the i-th ($1 \leq i \leq n$) index in the multi-index.
LP: Number of leaf level index pages.
NLP: Number of nonleaf level index pages.
np: Number of pages occupied by a record when the record size is larger than the page size.

We compare our multi-index index with the multiple cluster index with respect to the storage, retrieval, and update costs. As pointed out in [2], the parameters $k(i)$ ($1 \leq i \leq n$) that model the degree of reference sharing, impact the costs most significantly. Two objects share a reference if they reference the same object. To simplify our model we make the following assumptions.

- There are no partial instantiations. This implies that $D(i) = N(i+1)$; that is, each instance of a class $C(i)$ is referenced by instances of class $C(i-1)$.
- All key values have the same length. This implies that all nonleaf node index records have the same length.

Table 1: Parameters for Index Schemes Evaluation (in bytes)

UIDL = 8	kl = 8	kll = 2	rl = 2	nuid = 2	ol = 6
pp = 4	L = 14	L' = 14	f = 218	d = 146	P = 4096

- The value of attributes are uniformly distributed among instances of the class defining the attributes.
- For index structure we assume B-tree implementation.
- Attribute values may be multiple-valued.
- When an index record size is over a page, *the size of page pointers are ignored* to simplify the model. This is justifiable because the size of a page is about $4096bytes$ while a page pointer plus an object identifier is about $10bytes$.

All the assumptions except for the last two are the same as in [2], because we want to include multiple paths for a cluster to complement their study. For consistency we use the same values for the parameters as in [2] which are listed in table 1.

3.2 Storage Cost

The number of leaf node pages LP in a B-tree index is equal to the number of distinct index values divided by the number of index records fitting in one page.

$$LP = \lceil D(n)/\lfloor P/One\_Index\_Size \rfloor \rceil$$

Given the number of leaf node pages LP the number of nonleaf pages is evaluated level by level in a B-tree structure. Assume that the height of a B-tree is h. Let $LO = min(D(n), LP)$. The number of pages at each level of the B-tree is

- Level h (leaf): LO
- Level $h-1$: LO/f where f is the fanout from a node.

- Level $h-2$: LO/f^2 where f is the fanout from a node.

  ...

- Level 0 (root): 1.

Thus the number of nonleaf pages is

$$\begin{aligned} NLP &= LO/f + LO/f^2 + \cdots + LO/f^{log_f(LO)} \\ &= (LO/f - 1/f)/(1 - 1/f) \\ &= (LO - 1)/(f - 1) \end{aligned}$$

We use the formulae developed in [2]. In our study we observe the total disk pages as a function of the parameter k_1, k_2 and k_3 – reflecting the sharing of objects.

3.2.1 Single-valued attributes

Nested Index : The number of leaf pages is

$$LP = \lceil D(n)/P/XN \rceil$$

where XN the average size of leaf-node index record is

$$XN = k(1,n) * OIDL + kl + ol$$

Here $k(1,n)$ is the average number of instances of class $C(1)$ having the same value for the nested attribute $A(n)$.

$$k(1,n) = \prod_{i=1}^{n} k(i)$$

Path Index : The number of leaf pages is

$$LP = \lceil D(n)/P/XP \rceil$$

, where XP the average size of a leaf-node index record is

$$XP = PN * OIDL * n + kl + ol$$

, where

$$PN = \prod_{i=1}^{n} k(i)$$

Multi-index : For the n-th index in a multi-index the number of leaf pages is

$$LP(n) = \lceil D(n)/\lfloor P/XM(n) \rfloor \rceil$$

where $XM(n)$ the average size of a leaf-node index record is

$$XM(n) = k(n) * OIDL + kl + ol$$

For the i-th index ($1 \leq i < n$) the number of leaf node pages is

$$LP(i) = \lceil D(i)/\lfloor P/XM(i) \rfloor \rceil$$

where $XM(i)$ the average size of a leaf-node index record is

$$XM(i) = (k(i) + 1) * OIDL + ol$$

For the total number of pages we add all the disk pages of the n levels of the indices up. For our multi-index each level is a single cluster index.

Adjustment It is observable that compared with the *nested index* and *path index*, the *multi-index* have extra support for indexing attributes. For instance, in the *multi-index* in addition to the index support for nested objects, it supports single cluster index on end cluster of a path. Thus to compare *more accurately* among the three index schemes we need to add the storage cost of a single cluster index for the last cluster to the *nested index* and *path index*. In our case we need to add the $LP(3)$ of the *adapted index* to the storage cost of the *nested index* and *path index* for adjustment.

Further as pointed out in [2] that the *nested index* requires the support of *backward* traversal of objects for the update operations. For a path of length n, the structures for *backward* traversal of objects is equivalent to the structures of the first $n-1$ levels of a multi-index. Thus, if there are *no backward* links existed in the DBMS, we should add this overhead to the *nested index*; otherwise,

there exist backward links in the DBMS we can drop the storage cost for the first $n-1$ levels of a *multi-index*. In either case the adjusted *nested index* storage equals to that of the *multi-index* plus the pre-adjusted *nested index* storage. The *path index* does not require the support of backward links. Thus as a fair storage comparison for the three schemes we assume that there are no backward links pre-existed in the DBMS.

Based on the above discussion we evaluate the case of path of length three. Assume that all attributes are single-valued (i.e., $k(i) = N(i)/D(i)$) and the set of referenced objects equals to the set of objects of the referenced class (i.e., $D(i) = N(i+1)$ no partial instantiation). The total number of disk pages is

$$\begin{aligned} Total &= LP + \frac{LP-1}{f-1} \\ &= \frac{f*LP-1}{f-1} \end{aligned}$$

- For Multi-index: The total number of indices are the sum of the pages of all the levels.

  Level 1 :

$$\begin{aligned} LP(1) &= D(1)*XM(1)/P \\ &= \frac{N_1*s_1}{P}*(OIDL + \frac{OIDL+ol}{k_1}) \end{aligned}$$

  Level 2:

$$\begin{aligned} LP(2) &= D(2)*XM(2)/P \\ &= \frac{N_1}{P}*(\frac{OIDL}{k_1} + \frac{OIDL+ol}{k_1*k_2}) \end{aligned}$$

  Level 3:

$$\begin{aligned} LP(3) &= D(3)*XM(3)/P \\ &= \frac{N_1}{P}*(\frac{OIDL}{k_1*k_2} + \frac{kl+ol}{k_1*k_2*k_3}) \end{aligned}$$

- For *nested index* with $n = 3$:

$$\begin{aligned} LP &= D(3)*XN/P + Total\_LP\_of\_Multiindex \\ &= \frac{N_1}{P}*(OIDL + \frac{kl+ol}{k_1*k_2*k_3}) + Total\_LP\_of\_Multiindex \end{aligned}$$

- For *path index* with $n = 3$:

$$\begin{aligned} LP &= D(3) * XP/P + LP\_of\_Level3\_of\_Multiindex \\ &= \frac{N_1}{P} * (3 * OIDL + \frac{kl + ol}{k_1 * k_2 * k_3}) + LP\_of\_level3\_of\_Multiindex \end{aligned}$$

Analysis It is observable from the above formulae that:

- For all the three index schemes the total number of pages decreases as $k_1 * k_2 * k_3$ increases, i.e., the more sharing among objects, the smaller the size of the index.
- The *multi-index* always has *smaller* size than *nested index*.
- When the degree of sharing is high, for instance $k_1 * k_2 * k_3 > 1000$, the *path index* has a size slightly larger than that of the *nested index*.
- Between the *path index* and the *multi-index*, it depends on the degree of sharing of objects. When the sharing is low, for instance $kl = k_2 = k_3 = 1$, the *path index* has a size slightly smaller than the *multi-index*;
- However, when the sharing is high the *multi-index* is n times smaller size than the *path index*, where n is the length of a path.
- For the three schemes in terms of contributing to the total storage cost among k_1, k_2, k_3, k_1 is the most important one, followed by k_2, k_3.

Comparison Figure 5.(a) shows the comparison of storage cost for the indices when k_1 varies and $k_2 = k_3 = 1$. Figure 5.(b) shows the similar case with $k_2 = k_3 = 5$ and the instance number of class $C(1)$ equal to $N(1) = 20,000$. We observe that the *multi-index* has the lowest storage cost in general, except when there is no sharing of objects ($k = 1$) it has slightly larger size than that of the *path index*. The *nested index* also has a storage slightly lower than the *path index* when the degree of object sharing is high.

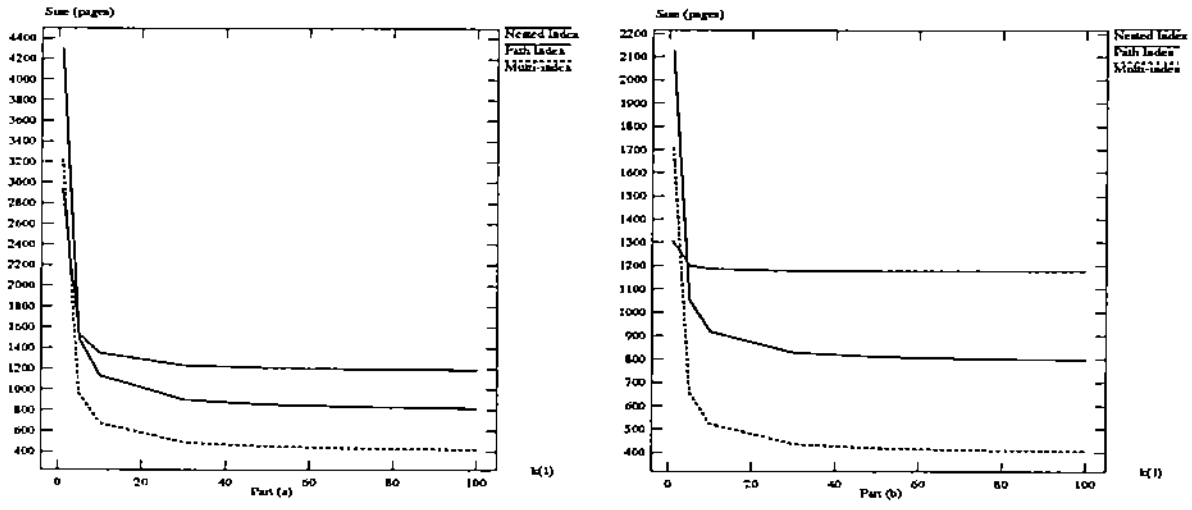

Figure 5: Part(a) shows the storage overhead for k(2)=k(3)=1; Part(b) shows the storage overhead for k(2)=k(3)=5. For both cases N(1)=20,000, and number of classes=3.

3.2.2 Multiple-valued Attributes

We consider that a pointer can have multiple values, i.e., a pointer can have different components pointing to different objects. For instance a "Student" can have a pointer "committees" which points different "Professor" objects. In Figure 6 we have class $C(i)$ pointing to class $C(i+1)$, each class has number of instances of $N(i)$ and $N(i+1)$ respectively. $D(i)$ is the number of distinct values for attribute $A(i)$. Each pointer has two values. We denote the number of multiple links for one point attribute as $s(i)$. We assume that there is no partial instantiation, i.e., every instance of class $C(i+1)$ is referenced by $C(i)$, thus $D(i) = N(i+1)$. To compute the index record size we need to evaluate $k(i)$ the average number of instances of class $C(i)$ that point to the same instance of class $C(i+1)$. We observe that from the class $C(i)$ point of view the number of links is $N(i) * s(i)$; from the class $C(i+1)$ point of view the number of links is $N(i+1) * k(i)$. They refer to the same set of links

$$N(i) * s(i) = N(i+1) * k(i)$$

Thus we have

$$k(i) = \frac{N(i)}{N(i+1)} * s(i)$$

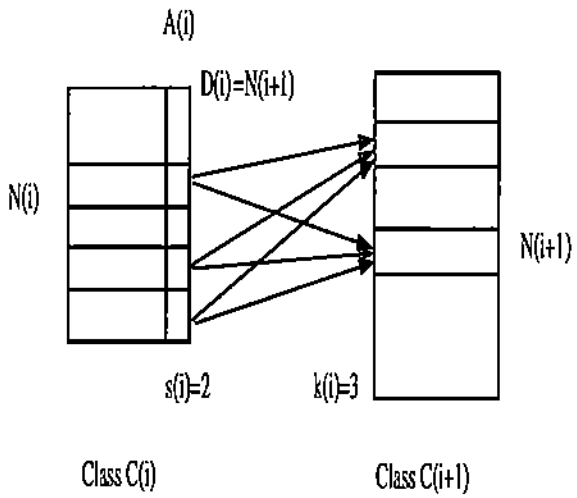

Figure 6: A new parameter s(i) to reflect a multiple-valued attribute.

$$= \frac{N(i)}{D(i)} * s(i)$$

Adjustment After the same adjustment as in the *single-valued* attribute case. We have the following storage costs for three schemes under the *multiple-valued* attribute assumption:

- For Multi-index: The total number of indices are the sum of the pages of all the levels.

  Level 1 :

$$\begin{aligned} LP(1) &= D(1) * XM(1)/P \\ &= \frac{N_1 * s_1}{P} * (OIDL + \frac{OIDL + ol}{k_1}) \end{aligned}$$

  Level 2:

$$\begin{aligned} LP(2) &= D(2) * XM(2)/P \\ &= \frac{N_1 * s_1 * s_2}{P} * (\frac{OIDL}{k_1} + \frac{kl + ol}{k_1 * k_2}) \end{aligned}$$

  Level 3:

$$LP(3) = D(3) * XM(3)/P$$

$$= \frac{N_1 * s_1 * s_2 * s_3}{P} * (\frac{OIDL}{k_1 * k_2} + \frac{kl + ol}{k_1 * k_2 * k_3})$$

- For *nested index* with $n = 3$:

$$\begin{aligned} LP &= D(3) * XN/P \\ &= \frac{N_1 * s_1 * s_2 * s_3}{P} * (OIDL + \frac{kl + ol}{k_1 * k_2 * k_3}) + Total\_LP\_of\_Multiindex \end{aligned}$$

- For *path index* with $n = 3$:

$$\begin{aligned} LP &= D(3) * XP/P \\ &= \frac{N_1 * s_1 * s_2 * s_3}{P} * (3 * OIDL + \frac{kl + ol}{k_1 * k_2 * k_3}) + LP\_of\_level3\_of\_Multiindex \end{aligned}$$

Comparison We compare the three index schemes for a path of length three. In Figure 7.(a) we set $N(1) = 20,000$, $s(1) = s(2) = 1$, $s(3) = 5$, $k(2) = k(3) = 5$ and vary $k(1)$. In Figure 7.(b) we set $N(1) = 20,000$, $s(1) = 5$, $s(2) = s(3) = 1$, $k(2) = k(3) = 5$ and vary $k(1)$.

In Figure 7.(b) when $k(1)$ is small the *multi-index* has a size much larger than both the *nested index* and *path index*. This is because in such cases the end classes have much smaller number of instances than the middle class; thus the *nested index* and *path index* have small size and the *multi-index* has large index for the middle class.

Analysis We observe from the formulae and the data that:

- For all the three index schemes the total number of pages increases as $s_1 * s_2 * s_3$ increases, i.e., the more multiple-values for an attribute, the larger the size of the index.
- The *multi-index* always has *smaller* size than *nested index*.
- For the *multi-index*, the first level size is proportional to s_1; the second level size is proportional to $s_1 * s_2$; the third level size is proportional to $s_1 * s_2 * s_3$.
- The sizes of the *path index* is proportional to $s_1 * s_2 * s_3$. When the degree of object sharing (k) is high, the *path index* is about n times as large as the *multi-index* where n is the length of a path.

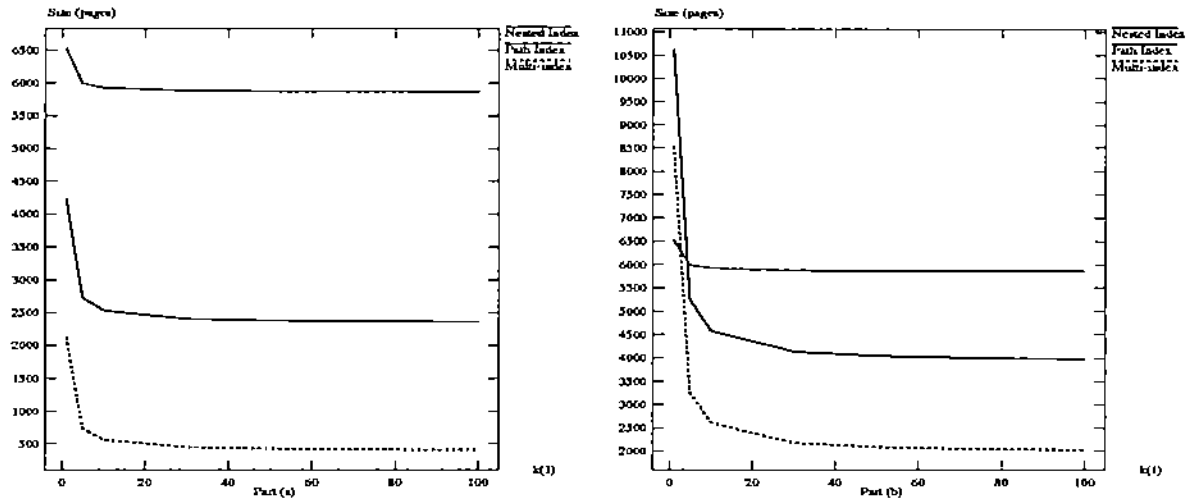

Figure 7: Part(a) shows the storage overhead for s(1)=s(2)=1, s(3)=5; k(2)=k(3)=5. Part(b) shows the storage overhead for s(1)=5, s(2)=s(3)=1; k(2)=k(3)=5. For both cases N(1)=20,000, and number of classes=3

- When s_3 is large and s_1 is small, the *multi-index* has *smaller* size than the *nested index*.
- When s_1 is large and s_3 is small the *multi-index* has a size *larger* than the *path index*.

Summary of Storage Costs

Contrary to the conclusion in [2] we found that

- The *nested index* alway costs more storage than the *multi-index* in any cases.
- For attributes having *single value* only, when the degree of object sharing is large (e.g., larger than five) the *multi-index* has the smallest storage cost, followed by *nested index* and *path index*; when the degree of object sharing is small (e.g., k=1 no sharing) the *path-index* has the smallest storage cost, followed by *multi-index* and *nested index*;
- For attributes having *multiple values*, only when the number of multiple values (s) is high towards the beginning classes of a path, the object sharing (k) high towards the end classes of the path, does the *path index* have less storage costs than both *multi-index* and *nested index*.

In all the other cases, both *multi-index* and *nested index* have lower storage costs than the *path index.*

3.3 Retrieval Cost

To obtain the set of object identifiers for a query involving nested attributes, both the *nested index* and *path index* requires only one index lookup, while for the *multi-index* n index lookups are needed, where n is the length of a path.

In comparing the three index schemes, the reference [2] making the assumptions that *backward pointers* for the updates of *nested index.* However, under this assumption the *multi-index* need *not* build the first $n-1$ levels of the indices, because those are for mapping the OID's and one can follow the backward pointers to do the same work. Thus *multi-index* can achieve *significantly better retrieval performance* than those reported in [2].

A single-key query is of the form $key = value$. A range-key query is of the form $value_1 < key < value_2$ or only one part of the range. We consider *single-key* retrieval queries to compare the three schemes. For *range-key* queries the results here can be scaled up by the number of values in the range. The basic cost model used here is the same as in [2]. We extend it to include *multiple-valued attributes* and four types of indices, namely, *nested index*, *path index*, and *multi-index with backward pointers*, and *multi-index without backward pointers.*

3.3.1 Multiple-valued Attributes

Single-Key Queries

Cost Model

- For the *nested index* and the *path index*, the retrieval cost consists of:

```
Pages of nonleaf nodes accessed + Pages of a leaf node
```

i.e. the number of index pages

$$A = h + \lceil \frac{X}{P} \rceil$$

where h is the number of nonleaf nodes of B-tree that must be traversed. Here h is the height of the B-tree

$$h = log_f LP$$

where LP is the number of leaf pages for the storage cost in section 3.2. X is the size of an index record; P is the page size.

- For a path of length of n, the *multi-index without backward pointers* the retrieval cost consists of:

```
Search of n-th index + Sum of page search of i-th index
```

In the nth index, we obtain a set of OID's S_n based on the index value. This retrieval cost is

$$A(n) = h(n) + \lceil \frac{X}{P} \rceil$$

where $h(n)$ is the height of the B-tree.

In the rest of the $n-1$ number of index lookups, the set of OID's S_n are mapped to S_{n-1}, ..., S_1, the set of OID's for retrieving the desired objects.

We denote the number of OID's of S_i as $NOID(i-1)$ for the input of the $(i-1)$th index lookup. We have

$$NOID(n-1) = k(n)$$

$$NOID(i) = k(i+1) * k(i+2) * ... * k(n), 1 \leq i < n-1;$$

By [11], the formula determines the number of pages needed when accessing k records randomly selected from a file containing n records grouped into m pages:

$$H(k,m,n) = m * [1 - \prod_{i-1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}]$$

Note that $H(k,m,n)$ is always less than k. Using this formula, the number of index leaf pages accessed in scanning the i-th index is

$$AL(i) = H(NOID(i), LP(i), D(i)), \quad if\ XM(i) \leq P$$

$$AL(i) = NOID(i) * \lceil \frac{XM(i)}{P} \rceil, \quad if\ XM(i) > P$$

To access each leaf page we need to traverse from the top of the B-tree. We assume the non-leaf pages are buffered. Thus, the number of pages accessed in the i-th index is

$$A(i) = H(NUID(i), LP(i), D(i)) + NLP(i), \quad if\ XM(i) \leq P$$

$$A(i) = NOID(i) * (\lceil \frac{XM(i)}{P} \rceil) + NLP(i), \quad if\ XM(i) > P$$

where $NLP(i)$ is the number of non-leaf pages for a B-tree evaluated in section 3.2.

- For a path of length of n, the *multi-index with backward pointers* the retrieval cost consists of:

```
Search of n-th index + Follow backward pointers
```

The $A(n)$ are the same as before. The $A(i)$ now becomes

$$A(i) = NOID(i)$$

assuming that in the worst case each node is in a different page.

Comparison For a path length of three we evaluate the retrieval costs for the four index schemes above. For each run we choose that the number of instances of objects in the first class $N1 = 200,000$; the degree of objects sharing $k(2)$, $k(3)$; and the number of multiple values for a pointer attribute $s(1)$, $s(2)$, $s(3)$ are fixed and only the value of $k(1)$ is *varied.*

The retrieval cost formulae for a path of length three:

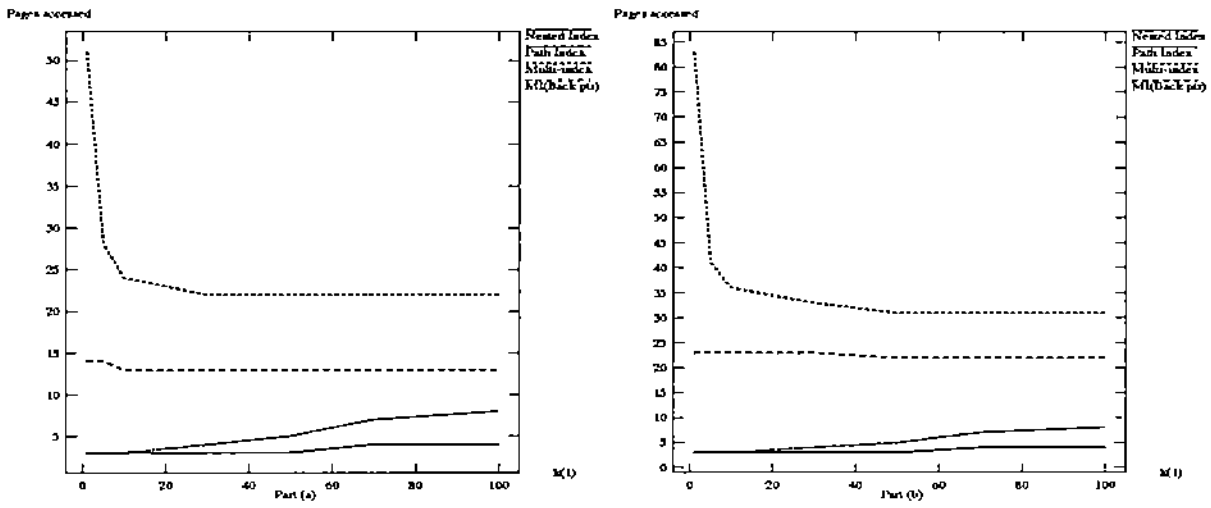

Figure 8: (a) Single-key retrieval for k(2)=10, k(3)=1; (b) Single-key retrieval for k(2)=1, k(3)=10. For both cases, N(1)=20,000, number of classes=3, number of multiple values s(1)=4, s(2)=s(3)=2

- *Nested Index*

$$A = log_f(\frac{N_1 * s_1 * s_2 * s_3}{P} * (OIDL + \frac{kl + ol}{k_1 * k_2 * k_3})) + \lceil \frac{k_1 * k_2 * k_3 * OIDL + kl + ol}{P} \rceil$$

- *Path Index*

$$A = log_f(\frac{N_1 * s_1 * s_2 * s_3}{P} * (3 * OIDL + \frac{kl + ol}{k_1 * k_2 * k_3})) + \lceil \frac{k_1 * k_2 * k_3 * 3 * OIDL + kl + ol}{P} \rceil$$

- *Multi-index without backward pointers*

$$A = A_3 + A_2 + A_1$$

where

$$A_3 = log_f(\frac{N_1 * s_1 * s_2 * s_3}{P} * (\frac{OIDL}{k_1 * k_2} + \frac{kl + ol}{k_1 * k_2 * k_3})) + \lceil \frac{k_3 * OIDL + kl + ol}{P} \rceil$$

$$A_i = NLP_i + AL_i, \quad i = 1,\ 2.$$

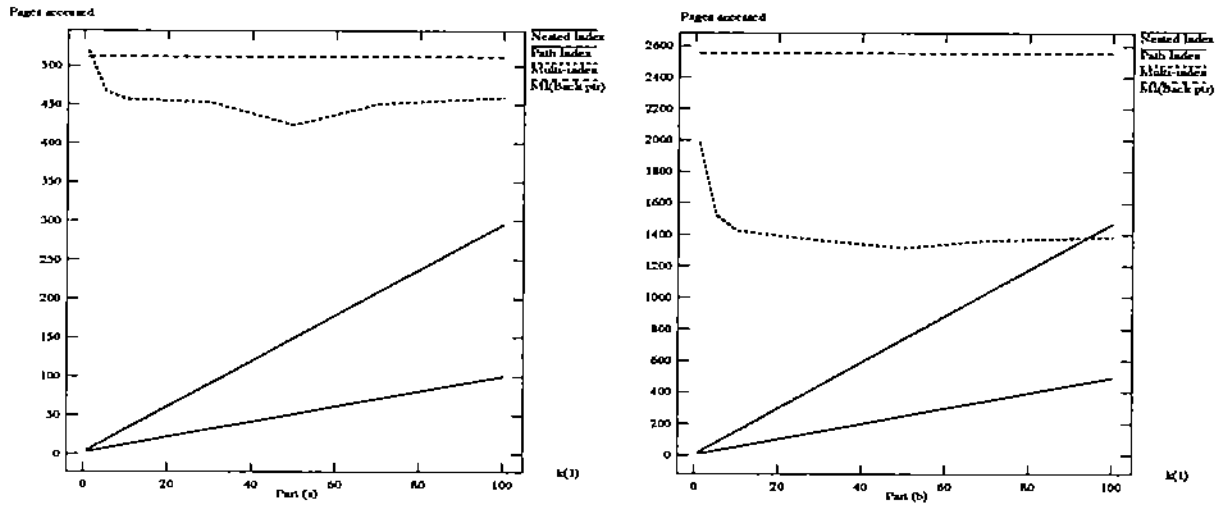

Figure 9: (a) Single-key retrieval for k(2)=50, k(3)=10; (b) Single-key retrieval for k(2)=k(3)=50. For both cases, N(1)=20,000, number of classes=3, number of multiple values s(1)=4, s(2)=s(3)=2

where $NLP_i = \frac{LP_i - 1}{f-1}$ and

$$LP_1 = \frac{N_1 * s_1}{P} * (OIDL + \frac{OIDL + ol}{k_1})$$

$$LP_2 = \frac{N_1 * s_1 * s_2}{P} * (\frac{OIDL}{k_1} + \frac{OIDL + ol}{k_1 * k_2})$$

$$AL_1 = k_2 * k_3 * \lceil \frac{k_1 * OIDL + kl + ol}{P} \rceil$$

$$AL_2 = k_3 * \lceil \frac{k_2 * OIDL + kl + ol}{P} \rceil$$

- *Multi-index with backward pointers*

$$A = A_3 + A_2 + A_1$$

where A_3 is the same as in the previous case. $A_2 = k_3$ and $A_1 = k_2 * k_3$

Figure 8.(a) compares the retrieval cost for the three indices for $s(1) = 4$, $s(2) = s(3) = 2$; $k(2) = 10$, $k(3) = 1$. Figure 8.(b) compares costs for $s(1) = 4$, $s(2) = s(3) = 2$; $k(2) = 1$, $k(3) = 10$.

Figure 9.(a) compares costs for $s(1) = 4$, $s(2) = s(3) = 2$; $k(2) = 50$, $k(3) = 10$. Figure 9.(b) compares cost for $s(1) = 4$, $s(2) = s(3) = 2$; $k(2) = 10$, $k(3) = 50$.

Analysis and Summary It is observable from the formulae and the Figure 8 and Figure 9 that:

- Of the three index schemes the *nested index* has the least retrieval costs, followed by the *path index* and the *multi-index*.
- The *nested index* and the *path index* has retrieval cost proportional to $k(1) * k(2) * k(3)$, once an index record size exceeds the page size. This can be observed from Figure 8 in that the two indices have the same costs when exchanging the values of $k(2)$ and $k(3)$.
- Both *multi-index* schemes (with/without backward pointers), the parameters, in the order of decreasing contribution to the cost, are $k(3)$, $k(2)$ and $k(1)$. They have a retrieval cost proportional to $k(1) + k(1) * k(2 + k(1) * k(2) * k(3)$, once an index record size exceeds the page size.
- The number of *multiple values* for an attribute $s(i)$ affects *multi-index* more than the *nested index* and *path index*. This is because in the first $n - 1$ indices of a path for the *multi-index* the *multiple* index lookups are needed, which in the worst case needs to load all the non-leaf nodes of the B-tree. The $s(1)$ contributes the most to the retrieval cost for the multi-index, followed by $s(2)$, and $s(3)$.
- In the two *multi-index* schemes, the one with the backward pointers outperforms the one *without* the backward pointers when the degree of object sharing towards the end of a path ($k(2)$ and $k(3)$) is low. The *opposite* is true when the degree of object sharing towards the end of a path ($k(2)$ and $k(3)$) is *high*.
- In Figure 9 for the *multi-index* without using backward pointers, we observe a minimum value for retrieval cost when $k(1)$ is the middle of a set of values. This is because before the index record size reaches the page size, the dominant cost for retrieval is the traversal of the non-leaf node B-tree; when the $k(1)$ increases the height of a B-tree decreases, thus we see a decrease

in retrieval cost. However, when the index record size exceeds the page size, the dominant term becomes the number of leaf nodes which increase with $k(1)$.

- When the index record size exceeds the page size the *multi-index* increase slower than the *path index* and *nested index* and when $k(1)$ exceeds a threshold value, the retrieval costs for the *path index* and *nested index* are greater than that of the *multi-index.*

- For the *multi-index* a more suitable index structure than B-tree index for mapping one set of OID's to another set of OID's maybe *hash index.* This is because for a B-tree index for each OID in the set we need to traverse the B-tree from top down; while in hash index it takes a near constant time lookup if a well-balanced hash function can be used. Also the mapping of OID's can be performed *parallelly* regardless of the index structures.

- For the update of *nested index*, reference [2] assumes that *backward pointers* are supported by the DBMS. We found that this condition can be dropped, instead the first $n-1$ indices in a *multi-index* of a path of length n, performs backward traversal of objects. Thus, the combination of *nested index* and *multi-index* is a feasible for supporting queries for nested attributes *without the requirement of backward pointers.*

3.4 Update Cost

We consider two kinds of updates: one is updating the values of the last class in a path; the other is changing the links on a path. We use the number of disk page access as the measure for update cost and B-tree as index structures.

- *Updating values of an indexed attribute of the last class in a path:*

  For a *single path* all the three schemes require the same updates to an index structure. For instance in a B-tree index, it is a delete of a node with the old index value, followed by an insert of the same node except that the index value is the new one.

- *Updating links on a path:*

As pointed out in [2] the *nested index* requires: a) two forward traversals to the end of paths based on the old link and the new link values to find the old and new index values; b) a *backward* traversal on the path to acquire the set of identifiers of objects with links to this objects on which the link is changed; c) locate the two index records and shift the set of OID's acquired, from the index record associated with the old link to that with the new link.

The *path index* requires only part a) and c) of the operations in the *nested index*. No backward traversals are needed because all the OID's on the path are stored in the leaf nodes of the path index. The *multi-index* only requires part c) of operations in the nested index.

It is observable that in addition to the updates to a *single path*, both the *nested index* and *path index* have to update *all* the nested indices and path indices leading to the this same class for both kinds of updates. Unfortunately the number of such indices can be *many without a limit*, i.e., both the *nested index* and *path index* do not *scale* well. As a sharp contrast the *multi-index* always requires updates to only one single cluster index regardless of the number of the *multi-indices* built on the same class in any kind of update, i.e., the *multi-index* is well *scalable*.

From the above analysis for updates the *multi-index* clearly outperforms the *nested index* and *path index* in every aspects. Furthermore, the motivation for using indices is to avoid *linearly traversing* data through links during selection of data. It is observable that for *nested index* and *path index* the traversal of data is partially shifted from the *selection* queries to *update queries*. The the *nested index* does more traversals than the *path index*.

We use the same cost model as in [2].

- U: Total update cost.
- CFT: Cost of a forward traversal.
- CBT: Cost of a backward traversal.
- CBM: Cost of a B-tree update.

The reference [2] has presented update studies for *single-valued attributes* for update costs for the three schemes. For path length of two it studied the case of updating the first class in a path.

For path length of three it studied the case of updating the last class in a path. Here we *extend* the model to include the case for attributes being *multiple-valued* as in the storage cost analysis. For *single-valued attributes* we first summarize the results in [2]. We then study the case of a path of length three, updating the *middle* cluster which complements the study in [2].

Cost Model

- *Nested Index*:

$$U = 2 * CFT + pdiff * (CBT + CBM)$$

  where $pdiff$ is the probability of two index values ($A(n)$) in the end class of the two paths are different. Note that when the two index values are the same the *nested index* need not be changed, thus *no* backward traversal and updates to index are needed.

- *Path Index*:

$$U = 2 * CFT + CBM$$

  Note that the update to the B-tree index is needed even if the two index values are the same, because the path are different and a path index record contains the object identifiers in the record.

- *Multi-index*:

$$U = CBM$$

  Note only single cluster index (B-tree) needs to be updated.

3.4.1 Multiple-valued Attributes

Assuming updating the an object in the i^{th} class of a path.

- *Nested Index*: For forward traversal there are $n - i$ clusters to travel. For the first one there are $s(i)$ number of objects; for the second cluster there are $s(i) * s(i+1)$ number of objects; etc. Recall that $s(i)$ is the average number of multiple links for one point attribute. For each object the physical address has to be determined before the object can be fetched, two I/O operations are needed. Thus,

$$CFT = 2 * \sum_{l=i}^{n-1} \prod_{j=i}^{l} k(l)$$

For backward traversal is the reverse of the forward traversal, except when reaching the second cluster from the start of a path, we can get the OID's of the first cluster directly from the backward pointers of the second class. Thus

$$\begin{aligned} CBT &= 2 * \sum_{k=2}^{i-1} \prod_{j=k}^{i-1} k(j), \quad if\ i > 2 \\ CBT &= 0, \quad otherwise \end{aligned}$$

For the probability $pdiff$ the two index attributes in the end class of a path ($A(n)$) are different, we observe that from the class n backward to class i in a path, each value in class n has $\prod_{j=i+1}^{n} k(j)$ objects in class i linking to it; in class i there are $D(i)$ number of distinct values. Thus the probability of the two index values are the same in the end class of a path is

$$\frac{\prod_{j=i+1}^{n} k(j) - 1}{D(i) - 1}$$

Thus,

$$\begin{aligned} pdiff &= 1 - \frac{\prod_{j=i+1}^{n} k(j) - 1}{D(i) - 1}, \quad if\ i < n \\ pdiff &= 1, \quad if\ i = n \end{aligned}$$

For cost of maintaining B-tree index, we traverse the B-tree once to find the leaf record. If the index record containing new index values happen to be on the same page as the one obtained,

no more leaf pages need to be fetched; otherwise another traversal of B-tree needed. Each traversal needs fetching h pages of non-leaf nodes of the B-tree, where h is the height of a B-tree, plus a read and a write for the leaf page. Suppose the probability of two index records are on the same leaf page is denoted as pl.

$$\begin{aligned} CBM &= (h+2) + pl * (h+2) \\ &= (h+2) * (1+pl) \end{aligned}$$

For pl similar to $pdiff$, we have

$$pl = 1 - \frac{P/XN - 1}{D(n) - 1}, \quad if\ XN < P$$

$$pl = 1, \quad if\ XN > P$$

Where P/XN is the number of records in a page.

The cost of for index updates for *delete* and *insert* of objects are the same. Both require one forward and one backward traversal plus on traversal of B-tree to the leaf node. Thus,

$$D = I = CFT + CBT + CBM$$

For $pdiff$ and pl with computed values outside [0, 1] we round them to the closer point.

- *Path Index*: The CFT is the same as in the *nested index*. For the CBM two traversals of B-tree are needed only when the two values of $A(n)$ are different and on the different leaf pages. Thus,

$$CBM = (h+2) * (1 + pdiff * pl)$$

For delete and insert of objects the cost for index updates is:

$$D = I = CFT + CBM$$

where CFT and CBM are the same as in the nested index.

- *Multi-index*:

  Only the B-tree for the i^{th} needs to be updated.

$$U = CBM = (h+2)*(1+pl)$$

  For delete and insert of objects the cost for index updates is:

$$D = I = h+2$$

Adjustment From discussion at the beginning of section 3.4, we know that the cost of both the *nested index* and *path index* should be multiplied by the number of indices built on a class in a path. Suppose that the average number of indices built on a class is m; U' as the adjusted cost; U is the cost of pre-adjusted cost. We have

$$Nested\ index:\ U' = m*U$$

$$Path\ index:\ U' = m*U$$

$$Multi-index:\ U' = U$$

For path of length three we develop the formulae for the update costs for the three schemes. We denote U_i for updates on the ith class of a path.

- *Nested Index*:

$$U_i = 2*CFT_i + pdiff_i*(CBT_i + CBM)$$

  where

$$\begin{aligned} CFT_1 &= 2*(s(1)+s(1)*s(2));\ \ CFT_2 = 2*s(2);\ \ CFT_3 = 0 \\ CBT_1 &= 0;\ \ CBT_2 = 0;\ \ CBT_3 = 2*k(2) \\ pdiff_1 &= 1 - \frac{k(2)*k(3)-1}{D(1)-1};\ \ pdiff_2 = 1 - \frac{k(3)-1}{D(2)-1};\ \ pdiff_3 = 1 \end{aligned}$$

and

$$\begin{aligned} h &= log_f(LP(3)) \\ &= log_f(\frac{D(3) * XN}{P}) \\ pl &= 1 - \frac{P/XN - 1}{D(3) - 1} \end{aligned}$$

Recall that $D(3)$, XN were evaluated in section 3.2 for storage cost.

- *Path Index*:

$$U_i = 2 * CFT_i + CBM_i$$

where CFT_i is the same as in *nested index*.

$$\begin{aligned} h &= log_f(LP(3)) \\ &= log_f(\frac{D(3) * XP}{P}) \\ pl &= 1 - \frac{P/XP - 1}{D(3) - 1} \end{aligned}$$

Recall that $D(3)$, XP were evaluated in section 3.2 for storage cost.

- *Multi-index*:

$$U_i = (h_i + 2) * (1 + pl_i)$$

where

$$pl_i = 1 - \frac{P/XM(i) - 1}{D(i) - 1}$$

and

$$\begin{aligned} h_i &= log_f(LP(i)) \\ &= log_f(\frac{D(i) * XM(i)}{P}) \end{aligned}$$

Recall that $D(i)$, $XM(i)$ were evaluated in section 3.2 for storage cost.

Note that setting $s(1) = s(2) = s(3) = 1$, we have the formulae for the update costs of *single-valued attributes*, which are consistent with those in [2].

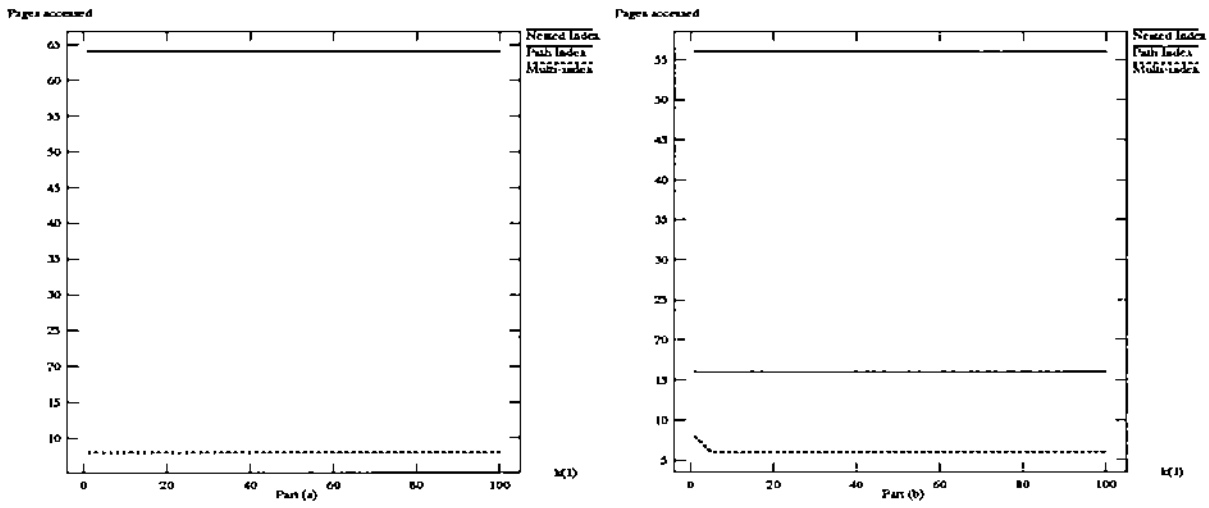

Figure 10: (a) Update for k(2)=10, k(3)=1; Updating class C(1). (b) Update for k(2)=10, k(3)=1; Updating class C(3). For both cases, N(1)=20,000, number of classes=3, number of multiple values s(1)=s(2)=s(3)=2, the number of indices m=2.

Comparison For a path length of three we evaluate the case for updates on the first, second and third classes on the path. For each case we choose that the number of indices built on nested objects to be $m = 2$, and the number of instances of objects in the first class $N1 = 200,000$. We fix the values of $k(2)$, $k(3)$, $s(1)$, $s(2)$, $s(3)$, and vary $k(1)$.

Figure 10.(a) compares the update cost for the three indices for $s(1) = s(2) = s(3) = 2$; $k(2) =$ 10,k(3)=1; $updating the class C(1). Figure\ 10.(b) compares costs for s(1)=s(2)=s(3)=2;k(2)=10, k(3) =$ 1; updating the class $C(3)$. Figure 11.(a) compares costs for $s(1) = s(2) = s(3) = 2$; $k(2) =$ 50,k(3)=10; $updating the class C(2). Figure\ 11.(b) compare scost for s(1)=s(2)=s(3)=2;k(2)=50, k(3) =$ 10; updating the class $C(3)$.

We can see that *multi-index* has the lowest cost among all the cases among the three schemes. We observe that *nested index* and *path index* have the same costs when updating the *first* and *second* class of a path. This is because in such cases *nested index* does not need to fetch pages to perform backward traversing, thus perform the same operations as *path index*.

Analysis We observe from the formulae and the computational data that:

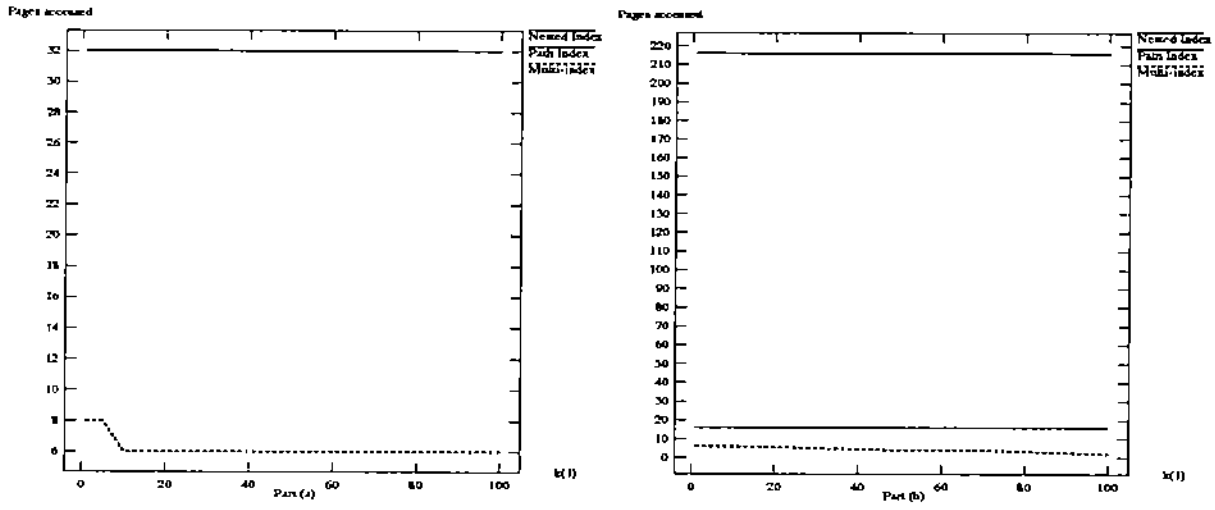

Figure 11: (a) Update for k(2)=50, k(3)=10; Updating class C(2). (b) Update for k(2)=50, k(3)=10; Updating class C(3). For both cases, N(1)=20,000, number of classes=3, number of multiple values s(1)=s(2)=s(3)=2, the number of indices m=2.

- Of the three index schemes the *multi-index* has the least update costs, followed by the *path index* and the *nested index.*

- *Multi-index*:

  The updates are single B-tree updates. The updates are proportional to the logarithm of the storage size of the class on which the index is built. Thus, based on our discussions on *storage cost*, it is observable that the updates costs increase logarithmically as the number of multiple links for one point attribute ($s(i)$) increases, because the number of leaf nodes for the index increases; the updates costs decrease logarithmically as the number of sharing of objects ($k(i)$) increases, because the number of leaf nodes decreases.

- *Nested index*: The update costs increase *linearly* as the number of nested indices built on a class in a path (m) increases. The forward traversal cost increases *linearly* as $s(i)$ increases; the backward traversal cost increases *linearly* as $k(i)$ increases; because more data objects need to be fetched. The update cost for B-tree index is similar to that of a *multi-index.*

- *Path index*: The update costs increase *linearly* as the number of path indices built on a class of a path (m) increases. The forward traversal cost is similar to that of the *nested index* only slightly higher. The update cost for B-tree index is similar to that of a *multi-index*.

Summary of Update Costs

From the above studies it is observable that

- The *multi-index* has the lowest costs of updates of the three schemes under *multiple-valued* attributes and *single valued* attributes assumptions.
- The *multi-index* is *scalable* and *flexible* in the sense that the update cost remains the same when the number of indices on the nested attributes with a common based attribute increases; and each common sub-path of multiple paths can share one single cluster index.
- Both the *nested index* and *path index* do not *scale* well in that the update cost *linearly increases* as the number of indices built on the nested attributes with a common based attribute increases.
- When updating the first two class on a path, the *nested index* and *path index* have the similar update costs in term o the number of disk pages accessed. This is because the assumptions of the existence of backward point for the nested index.
- When updating attributes on the end classes of a path, the *nested index* costs much more than the *path index* because the backward traversal of objects; the *path index* has a closer update cost to that of *multi-index* than other cases, because no forward traversal is required.
- Note that for the *nested index* update we assume the existence of *backward pointers*. If such a condition does not hold, one can use the *multi-index* structures to perform the backward mappings of the OID's, which are the same as those performed in *retrieval queries* using multi-index. However, the performance for the *nested index* updates is even worse, because it requires additional pages accessed.

Conclusion

In this paper we re-evaluates the three existing index schemes, namely nested index, path index, and multi-index for queries on nested attributes [2]. We propose a novel design for *multi-index* that allows reusing the existing single table index structures existed in a DBMS. This design of the *multi-index* has the following characteristics: 1) the same single table index structures can be used as the *dual* support for queries on nested objects across multiple tables, as well as on objects of a single table; 2) the update cost is the same as updating an index on a single table, regardless the number of indices built on the same attribute from different paths, i.e., it scales well for updates; 3) it does not require any traversal of objects during updates operations; and the storage cost is low compared with the other two schemes; 4) the retrieval cost is slightly higher than the *nested index* and *path index* when the degree of object sharing, the average number of objects pointing to the same object, is low; 5) when the index record size exceeds the page size the *multi-index* increase slower than the *path index* and *nested index*; when degree of object sharing is below the threshold values, the retrieval costs for the *multi-index* are much higher than those for the *path index* and *nested index*.

On the other hand, the *nested index* and *path index* have the following characteristics: 1) the index structures can be used for queries on nested objects across multiple tables only; 2) the updates require *traversals* of objects which incorporate heavy overheads compared with the *multi-index*; 3) the update costs increase linearly as the number of indices built on the same attribute from different paths, i.e., it does not scale well for updates; 4) they require traversals of objects during updates operations to the links; the nested index even requires backward traversals in addition to the forward traversals; it is observable that we introduce the indices to avoid the traversing objects which is of high cost and both *nested index* and *path index* shift this traversals from retrieval operations to the update operations; 5) they have low retrieval costs especially when the degree of object sharing is low, because one index lookup is needed compared with the multiple index lookups needed for the *multi-index*.

For the applications that *seldom* require updates *nested-index* and *path index* are favorite. For the applications that require *frequent* updates the *multi-index* is the best. For the applications that

lie between the two above, the *multi-index* is more promising in that it better balance the retrieval and update costs and reuse the existing index structures. We also found that *nested index* can be employed without requiring a DBMS to support backward pointers. The backward traversing can be achieved by using *multi-index.*

References

[1] R G G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[2] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), June 1989.

[3] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[4] Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, and Applications*, chapter 15, pages 371–394. ADDISON-WESLEY PUBLISHING COMPANY, 1989.

[5] S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases.* Springer LNCS 361, Heidelberg, 1989.

[6] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. *In Proc. International Workshop on Object-Oriented Database Systems, Asilomar (Ca.)*, September 1986.

[7] A. Kemper and G. Moerkotte. advanced query processing in object bases using access support relations. *In Proceedings of the 16th VLDB Conference*, 1990.

[8] A. Kemper and G. Moerkotte. Access support in object bases. *In Proceedings ACM SIGMOD International Conference on Management of Data*, 1990.

[9] O. Deux and et al. The O2 System. *Communicatons of the ACM*, 34(10), October 1991.

[10] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[11] S. B. Yao. Approximation Block Accesses in Database Organization. *ACM Communication*, 20:260–261, April 1977.